Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time

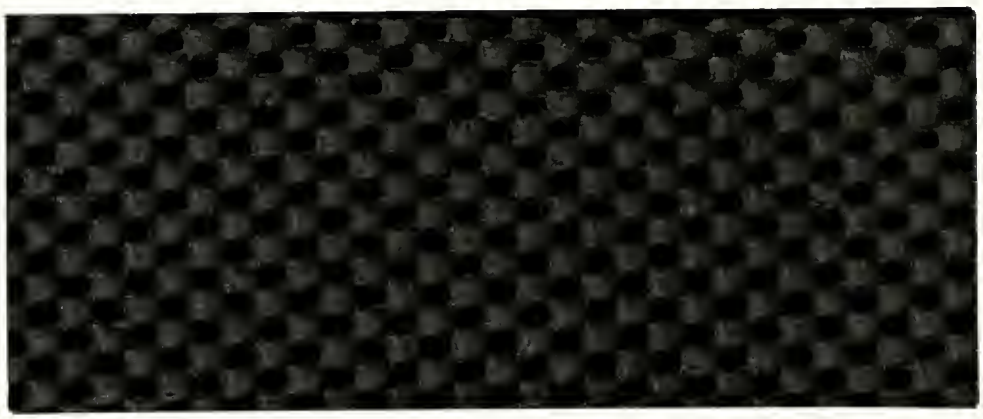
by

Richard Cole†
Uzi Vishkin‡

Ultracomputer Note #110
Computer Science Department Technical Report #244
October, 1986

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012



Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time

by

Richard Cole†
Uzi Vishkin‡

Ultracomputer Note #110
Computer Science Department Technical Report #244
October, 1986

† This research was supported in part by NSF grant DCR-84-01633 and by an IBM Faculty Development Award.

‡This research was supported in part by NSF grants NSF-DCR-8318874 and NSF-DCR-8413359, ONR grant N00014-85-K-0046 and by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077.

ABSTRACT

We define a novel scheduling problem; it is solved in parallel by repeated, rapid, approximate reschedulings. This leads to the first optimal logarithmic time PRAM algorithm for list ranking. Companion papers show how to apply these results to obtain improved PRAM upper bounds for a variety of problems on graphs, including: connectivity, biconnectivity, minimum spanning tree, Euler tour and st-numbering, and a number of problems on trees.

1. Introduction

The model of parallel computation used in this paper is a member of the parallel random access machine (PRAM) family. A PRAM employs p synchronous processors all having access to a common memory. In this paper we use an exclusive-read exclusive-write (EREW) PRAM. The EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes. See [Vi-83] for a survey of results concerning PRAMs.

Let Seq(n) be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form O(Seq(n)/p). A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*. A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

Most of the problems we consider can be solved by parallel algorithms that obey the following framework. Given an input of size n the parallel algorithm employs a *reducing* procedure to produce a smaller instance of the same problem (of size $\leq n/2$, say). The smaller problem is solved recursively until this brings us below some threshold for the size of the problem. An alternative procedure is then used to complete the parallel algorithm. We refer the reader to [CV-86d] where this algorithmic technique, which is called accelerating cascades, is discussed. Typically, we need to reschedule the processors in order to apply the reducing procedure efficiently to the smaller sized problem. Suppose the input for a problem of size n is given in an array of size n. A natural approach is to compress the smaller problem instance into a smaller array, of size $\leq n/2$. This is often done using a prefix sum algorithm (it takes O(log n) time on n/log n processors to compute the prefix sums for n inputs stored in an array). Thus if we need to reschedule the

processors repeatedly it is unclear how to achieve logarithmic time. Sometimes the rescheduling need not be performed very often: [CV-86a,C-86] show that for some problems (list ranking and selection) $\log^* n$ reschedulings suffice. Alternatively, one can use a fast random algorithm to perform the rescheduling, or at least an approximate rescheduling. (By approximate rescheduling we mean that we may not be able to partition the work evenly among the processors, but only approximately evenly.) Thus the need for rescheduling does not preclude O(log n) time optimal random algorithms. One of the main contributions of this paper is to provide an algorithm for performing approximate rescheduling deterministically in O(1) time. This is used to solve a novel scheduling problem. The solution to the scheduling problem leads to a logarithmic time optimal deterministic parallel algorithm for list ranking. A related rescheduling procedure is one of the tools that leads to a logarithmic time connectivity algorithm which is optimal unless the graph is extremely sparse.

We identify the following *duration-unknown* task scheduling problem. n tasks are given, each of length between 1 and e log n, e a constant; the total length of the tasks is bounded by cn, c a constant. (A task can be thought of as a program.) However, we do not know, in advance, the lengths of the individual tasks; in fact, they may vary, depending on the order of execution of the tasks. The problem is to schedule the n tasks on an EREW PRAM of n/log n processors so that the tasks are completed in O(log n) time; it is solved in Section 4.

We now discuss how to design algorithms that take advantage of this task scheduling algorithm. Given a problem, our job is to design a "protocol" for solving the problem by using a set of short tasks (each of length between 1 and e log n). This provides an important new opportunity for the designer of a protocol which is based on using the scheduling algorithm: the designer of the protocol need not know anything about the order of execution of the tasks. Such an opportunity for designing parallel tasks, without knowing in advance their lengths, with the guarantee that they will be scheduled efficiently, sounds very promising. However, this opportunity cannot be separated from a considerable difficulty in designing such a protocol: we have no control over the order of execution of the tasks, so we must ensure that the protocol works correctly regardless of the order of execution. We note that this style of protocol design may be useful for distributed systems that are not tightly synchronized; here too, we have to be sure that the protocol works correctly

regardless of the order of execution. Section 3 demonstrates how to design such a protocol for the following problem.

List ranking

Input: A linked list of length n. It is given in an array of length n, not necessarily in the order of the linked list. Each of the n elements (except the last element in the linked list) has the array index of its successor in the linked list.

The problem: For each element, compute the number of elements following it in the linked list.

Result: On the EREW PRAM, O(log n) time using n/log n processors.

Comments: 1. Wyllie [W-79] conjectured that it would be impossible to design a parallel logarithmic time list ranking algorithm using o(n) processors. Here, we are able to achieve a much stronger result than this.

2. The list ranking problem is often encountered as a subproblem in other parallel algorithms. The Euler tour technique on trees, which is given in [TV-85,Vi-85], consists of reducing a variety of tree functions into list ranking. [TV-85] used list ranking for reducing the biconnectivity problem into the connectivity problem. [CV-86b] gave recently a new "accelerated centroid decomposition (ACD)" parallel method for evaluation of tree expressions in logarithmic time using an optimal number of processors. The method was inspired by the important method of [MR-85] in an interesting way and provides new insights into the whole subject of parallel tree algorithms. The new ACD method performs a reduction of the tree expression evaluation problem to list ranking; the list ranking provides a schedule for evaluating the tree operations. The ACD method can be used to solve many of the problems considered in [MR-85]; generally, the ACD method seems to be simpler. For example, the ACD method evaluates tree expressions deterministically in logarithmic time using a linear number of operations. In comparison, [MR-85] achieved such a result for dynamically evaluating tree expressions, with a considerably more complicated randomized algorithm; they needed n processors to achieve logarithmic running time with a deterministic algorithm. The ACD method also generalizes to a family of problems for which there exists "leaf-to-root" serial algorithms. This family includes finding a minimum vertex cover and a minimum dominating set on trees. Recently, [SV-86] have shown how to apply the new list ranking algorithm for responding to queries regarding lowest common ancestors of pairs of vertices on tree.

Previous results: On the EREW PRAM, $O(\log n \log^* n)$ time using $n/(\log n \log^* n)$ processors [CV-86a]. Our new result is asymptotically better. However, it involves considerably larger constants. For all practical purposes the older result seems stronger. Previously [Vi-84] gave a randomized algorithm which runs in $O(\log n \log^* n)$ with overwhelming probability using $n/(\log n \log^* n)$ processors. [CV-86a] gives another list ranking algorithm, which for fixed k, runs in $O(k \log n)$ time using $n \log^{(k)} n / \log n$ processors.

Part 2 of this research (the paper [CV-86c]) shows how to apply the approximate scheduling method together with the new list ranking algorithm in order to derive improved PRAM upper bounds for a variety of problems on graphs, including: connectivity, biconnectivity, minimum spanning tree, Euler tour and st-numbering.

As can be seen, the results presented here improve on previous work in [CV-86a]. The main contributions of [CV-86a] were the deterministic coin tossing technique and a methodology for scheduling that used as few reschedulings as possible. While the details of their reschedulings were quite intricate, the rescheduling procedure itself was standard. We make two main contributions here. First, we provide a new approach to the rescheduling problem. Second, we show how to solve the list ranking problem in the novel framework imposed by our solution to the rescheduling problem. In addition, the companion papers show anew the central role played by the list ranking problem.

Historical remark. The goal of this remark is to clarify the relationship among the several joint publications of the authors. [CV-86a,d] are 'first generation' publications. [CV-86d] (in STOC 86) gives results on list ranking and graph connectivity; the final version of this first generation list ranking algorithm is given in [CV-86a]. [CV-86c,e] and the present paper are 'second generation' publications. [CV-86e] (in FOCS 86) gives second generation results on list ranking and graph connectivity; the final version of this list ranking algorithm is given in the present paper, and the final version of the work on connectivity from [CV-86d,e] is given in [CV-86c]. [CV-86b] extends the second generation list ranking algorithm to algorithms for tree problems.

In section 3 we give the new list ranking algorithm. In section 4 we present the duration-unknown task scheduling algorithm.

2. Preliminaries

We give below a useful and simple scheme, due to Brent, for designing parallel algorithms. Later, we discuss implications of this scheme for the formulation of complexity results regarding the performance of parallel algorithms.

Theorem (Brent). Any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented by p processors in time $\lceil x/p \rceil + t$.

Proof of Brent's theorem. Let x_i denote the number of operations performed by the algorithm at time i($\sum_{1}^{t} x_i = x$). We use the p processors to "simulate" the algorithm. Since all the operations at time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_{1}^{t} \lceil x_i/p \rceil \leq \sum_{1}^{t} (x_i/p + 1) \leq \lceil x/p \rceil + t .$$

□

Remarks. 1. Brent's Theorem is stated for parallel models of computation where not all computational overheads are taken into account. Specifically, the proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i at the beginning of time i in the algorithm. The second is to assign the processors to their jobs.

2. Often, in the present paper, it is straightforward to overcome the implementation problems posed by Brent's theorem without increasing the running time or the number of processors in order of magnitude. Therefore, we allow ourselves to switch freely from a result of the form "O(x) operations and O(t) time" to "x/t processors and O(t) time" (and vice versa, which is always correct). However, we avoided doing this where there are difficulties with these implementation problems.

3. List Ranking

We give an optimal O(log n) time algorithm to solve the following problem.

Input: A linked list of n nodes, stored in an array with index range [0: n−1]. For each node, we store the pointer and the distance to its successor in the arrays D(0: n−1) and R(0: n−1), respectively. For the last node v in the list we have R(v) = 0 and D(v) = **nil**.

Problem: Compute into array R, for each node u, the distance from u to the end of the list.

It is useful to assume that each node knows its predecessor in the list. This can be computed in O(1) time using O(n) operations, in the obvious way.

The algorithm starts with a series of O(log n) steps. Each step takes O(1) time. In each step we reduce the problem to a smaller subproblem by removing a set of non-adjacent nodes from the list. We remove node u, the successor of node v (that is u = D(v)), by the following pair of assignments.

R(v) := R(v) + R(u);

D(v) := D(D(v))

At each step of the algorithm each processor is associated with a node. If these assignments are performed by a processor associated with node v this is called a *traversal* by node v; if they are performed by a processor associated with node u this is called a *removal* by node u.

Our algorithm has three stages.

Stage 1. In O(log n) steps the input list will be reduced to a list of at most n/log n + 1 nodes. This will take O(log n) time and O(n) operations.

Stage 2. We compute the list ranking on the remaining list (the *reduced* list) using Wyllie's list ranking algorithm [W-79]; it will perform O(n) operations in time O(log n).

Stage 3. We show how to "reconstruct" the list ranking of nodes that were removed in Stage 1; it is reconstructed from the list ranking of the reduced list, computed in Stage 2. We first observe that given the list ranking for the list present at the end of a step of Stage 1, the ranking for the nodes present at the start of the step can be computed in O(1) time. Specifically, let the vector R_{actual} contain the list ranking. Let u be a node which was removed from the list at a step of Stage 1. Suppose that $R_{actual}(D(u))$ is known. (Note that both D(u) and R(u) cannot be changed by our algorithm after this removal). Then $R_{actual}(u)$ can be computed by $R_{actual}(u) := R(u) + R_{actual}(D(u))$. Second we apply the idea of backtracking. Specifically, each processor "revisits" the operations it performed at Stage 1 from the most recent to the earliest; at the time of revisiting the operation of removing node u it simply computes $R_{actual}(u)$. We refer the reader to [CV-86a] for a more detailed discussion of the "backtracking" procedure required. The time complexity

of Stage 3 is dominated by the time complexity of the forward steps of Stage 1.

So the total complexity of the algorithm is O(n) operations and O(log n) time.

The rest of this section is concerned only with the traversals and removals of Stage 1. The goal is to obtain a reduced list of at most n/logn + 1 nodes (the reduced list will include the first node of the input list). We call the nodes in the reduced list *full* nodes. The traversals and removals are performed by a set of $n-1$ tasks, associated with each node in the list (except the first node). Each task performs at most $2 \log n - 1$ steps of traversals and removals. Our main effort in this section is to show how to formulate these tasks so that we can use the duration-unknown task scheduling algorithm from Section 4 to schedule them. We need the following definition.

Definition. An r-ruling set of a linked list is a subset U of the nodes of the list such that

(i) No two nodes of U are adjacent in the list

(ii) If v is a node in the list, the next node from U in the list is at most r edges (links) distant from v.

We recall that there is a log n-ruling set algorithm that performs O(n) operations in O(1) time [CV-86a]. To make the paper self-contained we have described a slightly modified version of this algorithm in the Appendix. The algorithm has the following property: The first node in the list is always placed into the ruling set unless its successor is the last node in the list (i.e. the list contains exactly two nodes). We remark, that by definition, the ruling set contains the last node in the list. Also, the algorithm actually provides a *stronger result that we use below*: Suppose we assign a processor to each node in the list. Then, solely by looking at v, its three predecessors and four successors, the processor can determine in O(1) time whether v is in the log n-ruling set.

The task of v, for any node v in the list.

At the beginning, the task of node v will be *waiting*. At some step of Stage 1 a processor will be assigned to node v and the task will become *active*. The task will remain active until it is completed. Upon completing this task the processor will be able to determine whether v is in the reduced list (i.e. whether v is a *full node*) or not. If v is not in the reduced list then node v either: performs a removal, or "marks itself for removal", or is already removed.

On becoming active the task determines if node v has been removed from the present linked list. If so, in one step of Stage 1, the processor completes the task of node v without removing any node and v is not a full node. So, suppose that v is in the present list. The processor of v will complete its task with the decision that v is a full node if and when the following two conditions hold:

(i) Node v has performed at least log n traversals.

(ii) The successor of node v in the present list is not marked for removal.

Each step comprises three **synchronized** substeps.

Substep 1.

If node v has performed at least log n traversals and the successor of node v in the present list is not marked for removal

then the task is completed (without even proceeding to Substep 2); v is a full node.

Substep 2.

If the successor of v is a full node

then if the predecessor of v is not active

then v performs a removal; the task is completed; v is not a full node.

else mark v for removal; the task is completed; v is not a full node.

Substep 3. v belongs to a "chain" of presently active nodes; the chain is of length at least one and is followed by a node that is not full (either a node whose task is waiting or a node marked for removal).

Use the stronger version of the log n-ruling set algorithm to find whether v is in a log n-ruling set with respect to this chain. (Note: the last node in the chain is always placed in the ruling set. Specifically, in the trivial case, where the chain consists of a single node, this node is in the ruling set.)

If v is not in the ruling set and is not the first node in the chain

then mark v for removal; the task is completed; v is not a full node.

else if v is not in the ruling set and is the first node in the chain

then v performs a removal; the task is completed; v is not a full node.

else (* v is in the ruling set *) v performs a traversal.

Let us analyze the algorithm.

The reduced list comprises nodes that have traversed at least log n nodes each, plus the first node in the input list. This implies:

Lemma 3.1. The length of the reduced list is at most n/log n + 1.

Lemma 3.2. The task of each node includes at most 2log n − 1 traversals.

Proof. We need the following observation. Consider the time at which node u is marked for removal (but not removed). There must be a chain of length $x \leq \log n + 1$ nodes which ends at (and includes) u and satisfies the following: 1. The last x−1 nodes of this chain are marked for removal simultaneously. 2. The first node of the chain is active and has performed less than log n traversals.

Now consider node v that has performed at least log n traversals. After the time at which node v performed its log n-th traversal, its successors could have formed a chain of at most log n − 1 nodes marked for removal. This chain cannot grow while node v traverses the chain. Finally, Substep 1 implies that node v completes its task after traversing the $\leq \log n - 1$ nodes in this chain. Lemma 3.2 follows. □

Each removed node was subject to exactly one traversal or removal operation in one of the n−1 tasks. This implies the following corollary.

Corollary 3.1. There are n−1 tasks. Each task is of length O(log n) and the total length of the tasks is O(n). Further, a task, once active, remains active until it is completed.

Corollary 3.1 describes exactly the problem solved by the duration-unknown task scheduling algorithm of Section 3. Thus these tasks require O(log n) time and O(n) operations.

We have shown

Theorem 3.1. There is a EREW PRAM algorithm for list ranking that runs in O(log n) time using n/log n processors, which is optimal.

4. Approximate scheduling

We solve two dynamic scheduling problems: a *duration-unknown task* scheduling problem (which is encountered in the list ranking problem) and a *processor* scheduling problem (which is encountered in the the connectivity problem in the companion paper [CV-86c]). We start by stating the task scheduling problem, and then informally describe the processor

scheduling problem (as the precise description is somewhat involved we defer it to this companion paper where it is needed). Our solution for both problems require rapid rescheduling of the processors. The mechanism that provides each rescheduling is an *object redistribution* step. We define the *object redistribution* problem, which is solved by each application of the redistribution step. Then, we show how to solve the task scheduling problem using repeated applications of this object redistribution step, and finally we describe the object redistribution step itself. This step is based on a pseudo-random redistribution. The pseudo-randomness is achieved by using an expander graph.

The *duration-unknown task scheduling problem* (for short, the task scheduling problem) is the following. We are given n tasks. A task is a program that is to be run by one processor. The *length* of the task is its sequential running time (we will measure this as a specific number of O(1) time steps, which we call *real steps*). The lengths of individual tasks may be dependent on the scheduling; however, we are guaranteed that no task has length greater than e log n, and that the total length of all the tasks is bounded by c n, regardless of the scheduling, for some constants c and e. The problem is to execute these tasks on n/ log n processors in O(log n) time. We are allowed to schedule, and reschedule the tasks as we wish, so long as the scheduling occupies only O(log n) time.

For the *processor scheduling problem* we have p processors and q tasks, $q \leq p$. Here, several processors can cooperate effectively on a single task (we will not define a task precisely at this point). The problem is the following. Given an (uneven) distribution of the processors among the tasks produce a considerably more even distribution in O(1) time. We remark that the task scheduling problem will be solved by redistributing the tasks, while the processor scheduling problem appears to require the redistribution of processors. In fact, we will also solve this problem by task redistribution, as will be seen in [CV-86c].

Let us define the *object redistribution problem*. Initially, we are given r *objects*, partitioned among p *collections* of objects. We are also given one processor per collection. Loosely speaking, the problem is to redistribute the objects among the collections so that they are more evenly distributed. For a more precise description we need some definitions. $size_i$, the *size* of collection i, $1 \leq i \leq p$, is the number of objects in collection i; the *weight* of collection i is $size_i^2$, the square of its size. Let $W = \sum_{i=1}^{p} size_i^2$ be the total weight of all the collections and let s be the number of objects present currently ($s = \sum_{i=1}^{p} size_i$).

Define the *minimum weight* of s objects, $W_{s,m}$, to be $p\lceil s/p\rceil^2$, and let W_m be the minimum weight for the current set of objects. Let f and g be constants ($f \geq g \geq 298$, but specified more precisely at the end of this Section). If W is bounded by either f p or g W_m then the collections are said to be *balanced* (i.e. either there are few objects present, or the objects are roughly evenly distributed).

The *object redistribution problem* is the following. Each application of the *redistribution step* should satisfy three demands:

(1) If the collections are unbalanced, the problem is to reduce the total weight of the collections by a multiplicative constant factor in O(1) time.

In addition, even if the collections are balanced,

(2) The total weight must not be increased.

(3) The maximum number of objects in any one collection must never increase.

Data structure. The object redistribution step uses the following data structure. The objects are stored at the leaves of complete binary trees. Formally, suppose a collection has α objects. If $\alpha = 1$ then this single object forms a (complete binary) tree. Otherwise, suppose $2^i \leq \alpha < 2^{i+1}$, $i \geq 1$. Then, 2^i of these objects form a complete binary tree and the remaining $\alpha - 2^i$ objects recursively form complete binary trees. So, in each collection, the objects are stored in a set of complete binary trees, no two of the same size. We maintain the following pointers: for each internal node in such a tree, pointers to its leftmost and rightmost leaves; in addition, we keep the leaves in each tree in a linked list, in left to right order. Finally, for each collection, we keep its trees in a doubly linked list, in increasing order by size.

We solve the task scheduling problem as follows. We set $r = n$, $p = n/\log n$. The tasks are the objects. Initially, we distribute log n tasks to each collection. Each collection will never contain more than log n tasks. We perform $O(\log^{(2)}n)$ of the following stages.

(i) Each processor performs $\log n/\log^{(2)}n$ real steps on the tasks in its collection. This is easily done in $O(\log n/\log^{(2)}n)$ time for the links in the trees containing the tasks allow us to access each successive task in O(1) time, starting at the leftmost task (leaf) in the smallest tree). We then reformat the trees so that they are of the form assumed by the object redistribution step; we explain how to do this in $O(\log^{(2)}n)$ time in Remark 2, below.

(ii) Perform an object redistribution step. This requires O(1) time.

Lemma 4.1: After h $\log^{(2)}n$ stages there are at most fn/log n incomplete tasks, for some constant h.

Proof: Consider a single stage. If, at the start of part (ii) of the iteration the collections are not balanced then the weight is reduced by a constant factor. Since the initial weight is n log n, this can happen only $O(\log^{(2)}n)$ times, say at most $h_1 \log^{(2)}n$ times, for some constant h_1 (for at that point the weight will have been reduced to fn/log n; i.e. the collections are balanced). Thus, on the remaining iterations, at the end of part (i), the collections are balanced. That is, either the total weight is bounded by f n/log n (Case 1), implying that there are at most f n/ log n tasks remaining, or the weight is bounded by g W_m but not by f n/log n (Case 2).

We claim that in Case 2 at least $1/(16g) \cdot n/\log n$ of the collections are not empty. This is seen as follows. We first note that s, the number of objects present, is at least p (= n/log n) for otherwise we would have $W_m \leq p$. This would imply $g W_m \leq f p$, (remember $f \geq g$), and therefore contradicts the assumption of Case 2. Next, suppose x of the collections are non-empty. Then W, the total weight of all the collections, is at least $x \cdot \lfloor s/x \rfloor^2$; also $W \leq g W_m$ and $W_m = p \cdot \lceil s/p \rceil^2$. That is $x \lfloor s/x \rfloor^2 \leq g p \lceil s/p \rceil^2$, and since $s \geq p \geq x$, we have $s^2/4x \leq 4g s^2/p$, or $x \geq p/(16g)$.

For each non-empty collection, in part (i) of the stage, log n/$\log^{(2)}n$ real steps were performed on the tasks at hand. Thus, in Case 2, at least $n/(16g \log^{(2)}n)$ real steps were performed on the tasks in part (i) of this stage. Let $h = h_1 + 16gc$. We have shown that following h $\log^{(2)}n$ iterations there can be at most f n/ log n tasks remaining incomplete. □

The final stage: Distribute the remaining (at most fn/log n) tasks evenly among the processors (this is done using a standard prefix sum computation, as in [CV-86a], for instance, in a further O(log n) time using n/log n processors). Each processor will then complete its allotted tasks by performing at most f e log n more steps.

Thus the task scheduling problem can be solved in O(log n) time on n/ log n processors.

Remark 1: It is convenient, in the list ranking application, to ensure that having started one task, the processor completes it before beginning another. All we have to do is ensure that

we do not redistribute the tasks that processors are currently executing. This is easily done (see the distribution step below) by redistributing from the right end of trees. We will then never redistribute the current task, which is at the leftmost end of some tree.

Remark 2: We explain how to reformat the trees at the end of part (i), above. Let the task currently being processed be at leaf v of tree T. To ensure all the trees are complete and distinct we need to partition T into complete subtrees (recall that T is the smallest tree associated with the collection; thus we are guaranteed that if we create distinct sized complete binary trees from T then no two trees in the collection will have the same size and we get a proper set of complete binary trees). To partition T, we follow a path from v to the root of T. Each maximal right subtree that we encounter is made into a new tree. This traversal takes $O(\log^{(2)}n)$ time.

The rest of this section is devoted to the algorithm for object redistribution. We distinguish a sequence of *classes* of collections, from the lightest to the heaviest as follows.

Class 0 comprises those collections of size 0 or 1.

Class i, $i > 0$, comprises those collections whose size is in the range $[2^i, 2^{i+1})$.

The redistribution algorithm performs a cascade of redistributions of objects from heavier collections to lighter collections. As we will see, this provides the desired weight reduction.

Remark. A simpler idea would have been to redistribute objects from large collections to small ones directly, rather than through this cascade. Unfortunately, we did not find a way to implement this simple idea deterministically.

We will perform an object redistribution from one collection, in class i, to another collection, in class j, **only if** $j < i-1$. This guarantees that we achieve a constant multiplicative reduction in the total weight of the collections involved in this transfer, as will be shown later.

We note that we have little control over the distribution of objects at the start of the redistribution step, and hence we have little control over the distribution of collections among the classes. Nonetheless, for any class containing many collections, we want to be able to perform redistributions of objects from most of these collections to other collections

of considerably smaller size. This motivates us to consider expander graphs.

Definition: A bipartite graph $G = (V_1, V_2, E)$, with $|V_1| = |V_2|$, is a $(d, \epsilon, \frac{1-\epsilon}{\epsilon})$-expander graph if for any subset $U \subseteq V_1$, with $|U| \leq \epsilon |V_1|$, the set $N(U)$ of neighbors of vertices in U has size $|N(U)| \geq (\frac{1-\epsilon}{\epsilon})|U|$, and G has vertex degree d.

It is convenient to assume that d is a power of 2 (if not, we can add additional edges to G; this will not affect the expander property). Given any $\epsilon < 1$, it is known that there exist expander graphs for sufficiently large d (where d is a function of ϵ only). Further such an expander graph can be built in $O(\log |V_1|)$ time using $|V_1|/\log |V_1|$ processors, for each fixed ϵ, as we show in the following remark. (See [LPS-86,JM-85,GG-81] for results on expander graphs.)

Remark. For our construction we need $\epsilon = 1/36$. In other word, for $|U| \leq |V_1|/36$, we need that $|N(U)| \geq 35|U|$. We use an expander graph based on the construction described following Theorem 2 of [JM-85]. There, a different definition of expander graphs, which is due to Margulis [M-75], is given; the definition follows: A bipartite graph $K = (V_1, V_2, E)$ is an (n, k, δ) expander graph if $|V_1| = |V_2| = n$, the degree is k, and for each subset U of V_1, $|N(U)| \geq (1 + \delta(1 - \frac{|U|}{|V_1|}))|U|$. (Comment: actually, [JM-85] do not restrict the degree; they merely require $|E| \leq kn$; however, their constructions all obey this less liberal definition.) [JM-85] give an expander graph K with $k = 5$ and $\delta > 1/10$. >From this graph, using standard methods, we can obtain the expander graph G needed for our construction, as follows. We "iterate" the construction r times, for some constant r (which depends only on ϵ), creating graph H. Namely, the vertices of H comprise the disjoint vertex sets V_1, V_2, ..., V_{r+1}; we place a copy of the edges of K between V_i and V_{i+1}, for $1 \leq i \leq r$. G has vertex sets V_1 and V_{r+1}. If in H, vertices $v \in V_1$ and $w \in V_{r+1}$ are connected by a path of length r, then in G, v and w are joined by an edge. K has vertex degree 5; thus G has vertex degree at most 5^r. Finding the constant r is easy: For $|U| \leq 35/36 |V_1|$ the graph K satisfies that $|N(U)| \geq (1 + \frac{1}{10}\frac{1}{36})|U|$, implying $|N(U)| \geq (1 + 1/360)|U|$. It is easy to see that in G, $|N(U)| \geq \min\{35/36 |V_1|, (1 + 1/360)^r |U|\}$. Determining r is now straightforward. (A more careful argument provides a much tighter bound.) There are two more issues to be considered in order to adapt the construction of [JM-85] to our needs:

(1) [JM-85] provide expander graphs for which $|V_1| = |V_2| = m^2$, where m is an integer. We actually wanted an expander graph on n vertices, not on m^2 vertices. So let $(m-1)^2 < n \leq m^2$. Instead of using an expander graph on n vertices, we use the expander graph on m^2 vertices. This implies the algorithm assumes m^2 processors are available; but such an algorithm is readily simulated on n processors, slowing it down by a factor of at most 2.

(2) We comment on the complexity of constructing the edges of K (and implicitly of G). Each vertex has degree five. Each edge can be readily determined in O(1) time by a single processor.

The object redistribution algorithm follows. For each collection we create two nodes: a *giving* node and a *receiving* node (we will also refer to these as the *giving* collection and the *receiving* collection). We connect the giving and receiving nodes by a $(d, \epsilon, \frac{1-\epsilon}{\epsilon})$-expander graph, the giving nodes being vertex set V_1. We perform the following object redistribution. Suppose there is an edge connecting the giving collection C_1 to the receiving collection C_2. Let C_1 and C_2 be in classes i_1 and i_2, respectively. If $i_1 > i_2 + 1$, then a number of objects are removed from C_1 and added to C_2. More precisely, let β be the number of leaves in the largest tree of objects in C_1 ($\beta = 2^{i_1}$); if $\beta \geq 8d$ then the number of objects transferred is $\beta/8d$; otherwise, no objects are moved.

The object transfer can be performed in O(1) time since it only involves manipulating the top O(log d) levels of the largest tree in C_1 (followed by a constant amount of tree reconstruction in C_1 and C_2). More precisely, for each collection with at least 8d objects, we divide the largest tree in the collection (of size $\beta \geq 8d$) into 8d equal sized subtrees. Up to d of these trees are transferred in the object redistribution (the rightmost trees are transferred in accord with remark 1, above). The remaining trees of size $\geq \beta/8d$ in the collection are then repeatedly paired together (only equal sized trees are paired), until there is at most one tree of each size. Since this involves at most 8d + log 8d trees it takes O(1) time using one processor per collection. The pairing process is then repeated with the trees that are transferred to the collection. There are at most d + log 4d trees involved in the latter pairing; so it also takes O(1) time using one processor per collection. We note that care must be taken to maintain the pointers for the tree nodes as the trees are manipulated; however, this is straightforward.

We proceed to show that the redistribution we have just described never increases the total weight, and when the collections are unbalanced, reduces it by a multiplicative constant. We also note that the redistribution never increases the size of the largest collection.

For the purposes of the analysis, we consider the redistributions to be performed in the following order (in fact, they are performed simultaneously). We consider each redistribution to be an ordered pair comprising the weight of the receiving collection followed by the weight of the giving collection. The redistributions are ordered lexicographically, and are performed in this order. Thus the redistributions into the lightest receiving collection are performed first; among these redistributions, the one from the lightest giving collection is performed first.

Claim. Let s_g be the initial size (that is, before the present redistribution step starts) of giving collection C_g and let s_r be the initial size of receiving collection C_r. Suppose that there is a redistribution from C_g to C_r. Following this redistribution, C_g has size at least $7/8\ s_g$, and C_r has size at most $s_r + 1/8\ s_g$.

Lemma 4.2: The weight reduction produced by the redistribution of the claim is at least $s_g^2/32d$.

Proof: Let $7/8s_g + a + b$ be the size of C_g immediately before the redistribution, let $s_r + c$ be the size of C_r immediately before the redistribution, and let a be the number of items transferred. We know that: (1) $s_r < s_g/2$ (since if s_g belongs to some class i then s_r can belong only to a class j, where $j \leq i-2$). (2) $c + a \leq 1/8\ s_g$. This follows from the above claim regarding the size of C_r). (3) $a + b \geq 0$ (this is trivial). And, (4) $16da > s_g$ (since a, the number of elements transferred is $> s_g/16d$). The weight reduction is given by:

$$(7/8\ s_g + a + b)^2 + (s_r + c)^2 - (7/8\ s_g + b)^2 - (s_r + c + a)^2 = 2a\ (7/8\ s_g + b - s_r - c).$$

Observe that (2) and (3) above imply that, $b-c = -((c+a) - (b+a)) \geq -s_g/8$. Using this and (1) above we get,

$$\geq 2a(7/8\ s_g - s_g/2 - 1/8\ s_g) = \frac{as_g}{2}.$$

By (4) above we get,

$$\geq \frac{s_g^2}{32d}\ \square$$

Corollary 4.1: Any sequence of redistributions reduces the total weight.

Corollary 4.2: Let W_g be the total weight of the giving collections, counting multiplicities, in a sequence of redistributions. Then the weight reduction for these collections, due to the redistributions, is at least $W_g/32d$.

In the rest of this section, we assume that the collections are unbalanced. We will show that *the weight of the giving collections, counting multiplicities, is a constant fraction of the total weight of the collections*. It follows, by Corollary 4.2, that the total weight of the collections is reduced by a multiplicative constant.

Let S_i be the set of collections in class i. Let $L_i = \bigcup_{k \geq i} S_i$. Let $|S_i|$ (resp. $|L_i|$) denote the number of collections in S_i (resp. L_i), and let $wt(S_i)$ (resp. $wt(L_i)$) denote the sum of the weights of the collections in S_i (resp. L_i). On the average there are s/p objects in each one of the p collections. Define the variable average to be $\lfloor \log_2 s/p \rfloor$. We refer to class average as the average class. Let α denote the smallest $i >$ average$+2$ such that $|S_i| > |S_{i-1}|/32$.

We define a set S_i, $i \geq \alpha$, to be *giving* if there are at least $|S_i|$ giving collections in S_i, counting multiplicities. We implement the expander graph so that whenever $|S_i| \geq 1/32|S_{i-1}|$ and $|S_i| \geq |L_{i+1}|$ then S_i is giving.

Our presentation proceeds as follows. First, we show that such an α exists. Second, we show that the total weight of the collections in L_α is a constant fraction of the total weight. Third, we show that the weight of the giving sets is a constant fraction of the weight of L_α. Finally, we show that the weight of the giving collections, counting multiplicities, in a giving set, is a constant fraction of the weight of the giving set.

Lemma 4.3. α exists.

Proof. We assume, in contradiction, that α does not exist and show that the collections must have been balanced, which is contrary to our assumption. To show this we observe the following:

(a) The total weight of the collections in classes 0,..,average-1 is less than W_m.

(b) $wt(S_{average}) \leq 4W_m$; $wt(S_{average+1}) \leq 16W_m$; $wt(S_{average+2}) \leq 64W_m$.

(c) $wt(S_i) \leq wt(S_{i-1})/2$ for $i >$ average$+2$ (since otherwise α would exist).

The total weight is therefore,

$$\sum_{i < \text{average}} wt(S_i) + wt(S_{\text{average}}) + wt(S_{\text{average}+1}) + wt(S_{\text{average}+2}) + \sum_{i > \text{average}+2} wt(S_i)$$

Observations (a),(b) and (c) above imply that this is

$$\leq W_m + 4W_m + 16W_m + 64W_m + 64W_m = 149W_m$$

Since $g > 298$ we conclude that the collections are balanced. □

Using a similar proof we can show that

Lemma 4.4. $wt(L_\alpha) \geq (g - 149)W_m$.

Corollary 4.3. The weight of L_α is a constant fraction of the total weight W. Specifically, $wt(L_\alpha)$ is at least W/2.

Henceforth, we only consider sets S_i with $i \geq \alpha$. We need to classify these sets according to whether they must contain many giving collections. Thus, we define a set S_i to be *large* if $|S_i| \geq |S_{i-1}|/32$. Likewise, it is *small* if $|S_i| < |S_{i-1}|/32$. We will show that the weight of the large sets is at least half the weight of L_α. The neighbors, for giving collections, will be provided by the edges of an expander graph. Thus if $|L_{i+1}|$ is large compared to $|S_i|$, we will not be able to guarantee that there are many giving collections in S_i. So we define S_i to be *useful* if $|S_i| \geq |L_{i+1}|$. We will show that the weight of the useful large sets is at least one fifth of the weight of the large sets. Finally, by choosing the right constant ϵ for the expander graph, we will ensure that each useful large set is a giving set.

Lemma 4.5. The weight of the large sets is at least 1/2 $wt(L_\alpha)$.

Proof. Let S_i, S_{i+k} be large sets and suppose that for each j, $1 \leq j < k$, S_{i+j} is not large. Then, as in the proof of Lemma 4.3, we can show that $wt(S_i) \geq \sum_{j=1}^{k-1} wt(S_{i+j})$. Let S_h be the large set of greatest index; we can also show that $wt(S_h) \geq \sum_{j \geq 1} wt(S_{h+j})$. Since S_α is large the lemma follows. □

Lemma 4.6. The weight of the useful large sets is at least 1/5 of the weight of the large sets.

Proof. Observe that the large set whose index is maximal must be useful. Consider a sequence of sets $S_{i_{k+1}+1}, S_{i_{k+1}+2}, \ldots, S_{i_0}$, which satisfies the following: (1) S_{i_0} is the only

set in the sequence which is both useful and large. (2) Either $S_{i_{k+1}}$ is both useful and large; or i_{k+1} is $\alpha-1$. (3) S_{i_k}, $S_{i_{k-1}}$, ..., S_{i_1}, S_{i_0} is the subsequence of large sets in this sequence. We show that $\sum_{j=1}^{k} wt(S_{i_j}) \leq 4\ wt(S_{i_0})$. Since any large set must lie in such a sequence the lemma follows.

Let T_{i_j} be the union of the small sets between S_{i_j} and $S_{i_{j-1}}$. Let the sequence $R_{i_{h+1}}$, R_{i_h}, ..., R_{i_0}, comprise the merge of the sequence S_{i_k}, ..., S_{i_0}, and the non empty sets in the sequence T_{i_k}, ..., T_{i_1}. We actually prove that $\sum_{j=1}^{h} wt(R_{i_j}) \leq 4\ wt(R_{i_0}) = 4\ wt(S_{i_0})$. This result is an immediate consequence of the following claim: *Claim 1*: For $1 \leq j \leq h$, $wt(R_{i_j}) \leq 2^{2-2j} wt(R_{i_0})$. This in turn, is an immediate consequence of the following two claims: *Claim 2*. For $1 \leq j \leq h$, $|R_{i_j}| \leq 2^j |R_{i_0}|$. *Claim 3*. For $1 \leq j \leq h$, the weight of any collection in R_{i_j} is at most 4^{2-2j} of the weight of any collection in R_{i_0}. Claim 3 is immediate.

Proof of Claim 2. For each set R_{i_j}, $j \geq 1$, we prove the following *assertion*: the number of collections in $\bigcup_{x=0}^{j-1} R_{i_x} \cup L_{i_0}$ is greater than the number of collections in R_{i_j}. Claim 2 follows by induction on j, from the assertion, when we note that $|L_{i_0}| \leq |R_{i_0}|$. The proof of the assertion breaks into two cases.

Case 1: R_{i_j} is a set S_{i_y}. Because S_{i_y} is not useful $|S_{i_y}| < |L_{i_y+1}|$, and the assertion follows.

Case 2: R_{i_j} is a set T_{i_x}. We note that $|T_{i_x}| < 1/31\ |S_{i_x}|$. Also, since S_{i_x} is not useful, $|S_{i_x}| < |T_{i_x}| + |L_{i_{x-1}}|$. Thus $|T_{i_x}| < 1/31\ (|T_{i_x}| + |L_{i_{x-1}}|)$. We deduce that $|T_{i_x}| < |L_{i_{x-1}}|$. Again, the assertion follows. □

We would like the giving collections in a useful large set S_i to have many receiving neighbors (which we call the neighbors of S_i, $N(S_i)$) in the expander graph in the set $\bigcup_{j=0}^{i-2} S_j$. So we choose $\epsilon = 1/36$ for the expander graph and show:

Lemma 4.7. A useful large set S_i has at least $|S_i|$ neighbors in $\bigcup_{j=0}^{i-2} S_j$.

Proof. We note that $|S_{i-1}| + |S_i| + |L_{i+1}| \leq 34|S_i|$ for S_i large and useful. There are two possibilities:

Possibility 1. $|S_i| \leq p/36$. Then S_i has at least 35 $|S_i|$ neighbors, of which at least $|S_i|$ lie outside $S_{i-1} \cup S_i \cup L_{i+1}$.

Possibility 2. $|S_i| > p/36$. Then S_i has at least 35p/36 neighbors. Observe that $|\bigcup_{j=0}^{i-2} S_j| \geq p/2$ (since i > average+2). Therefore, at least $\frac{17}{36}p$ of these neighbors are in $\bigcup_{j=0}^{i-2} S_j$. Also, i > average+2, implies that $|S_i| \leq p/4$. Thus S_i has at least 17/9 $|S_i| > |S_i|$ neighbors in $\bigcup_{j=0}^{i-2} S_j$. This completes the proof of Lemma 4.7.

Lemma 4.8. The weight of the giving collections is at least $1/2 \cdot 1/2 \cdot 1/5 \cdot 1 \cdot 1/4) > 1/2^7$ of the total weight W.

Proof. In Corollary 4.3 we showed that $wt(L_\alpha)$ is at least W/2. In Lemma 4.5 we showed that the weight of the large sets was at least 1/2 $wt(L_\alpha)$. In Lemma 4.6 we showed that the weight of the useful large sets was at least 1/5 of the weight of the large sets. In Lemma 4.7 we showed that each useful large set S_i has at least $|S_i|$ giving collections, counting multiplicities. Finally, since the size of the collections in each useful large set S_i lie in the range $[2^i, 2^{i+1})$, we conclude that the giving collections in S_i, counting multiplicities, have weight at least 1/4 $wt(S_i)$. The result now follows.

In the definition of balanced, above, we set g = 298, and f = max{g, 8d}. We have then shown:

Theorem 4.1: If the collections are not balanced then, in O(1) time, the redistribution algorithm reduces the total weight by a multiplicative factor of at least $1 - \frac{1}{32d}\frac{1}{2^7} \geq 1 - \frac{1}{2^{12}d}$.

5. References

[B-74] R.P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM* 21,2 (1974), 201-206.

[C-86] R. Cole, "An optimal parallel selection algorithm", Courant Institute technical No. 209, 1986.

[C-86a] R. Cole, "Parallel merge sort", to be presented, *Twenty Seventh Annual Foundations of Computer Science Conf.*

[CV-86a] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control* 70, 32-53. (1986), 32-53.

[CV-86b] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time", Courant Institute technical report No. 242, 1986.

[CV-86c] R. Cole and U. Vishkin, "Approximate parallel scheduling. Part II: applications to optimal parallel graph algorithms in logarithmic time", in preparation.

[CV-86d] R. Cole and U. Vishkin, "Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms", *Proc. Eighteenth Annual ACM Symp. on Theory of Computing*, 206-219.

[CV-86e] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems", *Proc. Twenty Seventh Annual Symp. on Foundations of Computer Science*, 1986.

[GG-81] O. Gabber and Z. Galil, "Explicit constructions of linear sized superconcentrators", JCSS 22(1981), 407-420.

[JM-85] S. Jimbo and A. Maruoka, "Expanders obtained from affine transformations", *Proc. Seventeenth ACM Symp. on Theory of Computing, 88-97.*

[LPS-86] A. Lubotzky, R. Phillips and P. Sarnak, "Ramanujan conjecture and explicit constructions of expanders and super-concentrators", *Proc. Eighteenth Annual ACM Symp. on Theory of Computing*, 240-246.

[M-75] G.A. Margulis, "Explicit constructions of concentrators", Prob. Per. Infor. 9(4), 1973, 71-80, English translation in Problems of Infor. Trans. 1975, 325-332.

[MR-85] G. Miller and J. Reif, "Parallel tree contraction and its application", *Proc. Twenty Sixth Annual Symp. on Foundations of Computer Science*, 478-489.

[Pi-86] N. Pippenger, "Sorting and selecting in rounds", manuscript.

[R-86] J. Reif, "An optimal parallel algorithm for integer sorting", *Proc. Twenty Sixth Annual Symp. on Foundations of Computer Science, 496-503.*

[SV-86] B. Schieber and U. Vishkin, "Parallel computation of lowest common ancestor in trees", in preparation.

[TV-85] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. of Comput.*, 14,4(1985), 862-874.

[Vi-83] U. Vishkin, "Synchronous parallel computation - a survey", TR 71, Dept. of Computer science, Courant Institute, NYU, 1983.

[Vi-84] U. Vishkin, "Randomized speed-ups in parallel computation", *Proc. Sixteenth Annual ACM Symp. on Theory of Computing*, 230-239.

[Vi-85] U. Vishkin, "On efficient parallel strong orientation", *Information Processing Letters* 20 (1985), 235-240.

[W-79] J.C. Wyllie, "The complexity of parallel computation," TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

6. Appendix: The log n-ruling set algorithm.

Assumptions about the input representation: The vertices are given in an array of length n. The entries of the array are numbered from 0 to $n-1$. The numbers are represented as binary strings of length $\lceil \log n \rceil$. We refer to each binary symbol (bit) of this representation by a number between 0 and $\lceil \log n \rceil - 1$. The rightmost (least significant) bit is called bit number 0 and the leftmost bit is called bit number $\lceil \log n \rceil - 1$. Each vertex has a pointer to the next vertex in the list (representing its outgoing edge). For simplicity we assume that log n is an integer**.

Here is a verbal description of an algorithm for the log n-ruling set problem. The algorithm is given later. Processor i, $0 \leq i \leq n-1$, is assigned to entry i of the input array (for simplicity, entry i is called vertex i). It will attach the number i to vertex i. So, the *present* "serial" number of vertex i, denoted $SERIAL_0(i)$, is i. Next, we attach to vertex i a new serial number, denoted $SERIAL_1(i)$, as follows. For each vertex i that is not the last vertex in the list, let i_2 be the vertex following i. Let j be "the index of the rightmost bit in which i and i_2 differ". Processor i assigns j to $SERIAL_1(i)$.

Example. Let i be ...010101 and i_2 be ...111101. The index of the rightmost bit in which i and i_2 differ is 3 (recall the rightmost bit has number 0). Therefore, $SERIAL_1(i)$ is 3.

A remark in [CV-86a] explains how j can be computed by a constant number of standard operations.

Next, we show how to use the information in vector $SERIAL_1$ in order to find a log n-ruling set.

Fact 1: For all i, $SERIAL_1(i)$ is a number between 0 and $\log n - 1$ and needs only $\lceil \log\log n \rceil$ bits for its representation. For simplicity we will assume that loglog n is an integer.

Let i_1 and i_2 be, respectively, the vertices preceding and following i. $SERIAL_1(i)$ is a local minimum if $SERIAL_1(i) \leq SERIAL_1(i_1)$ and $SERIAL_1(i) \leq SERIAL_1(i_2)$. A local maximum is defined similarly.

**The base of all logarithms in the Appendix is 2.

Fact 2: The number of vertices in the shortest path from any vertex in G to the next (vertex that provides a) local extremum (maximum or minimum), with respect to $SERIAL_1$, is at most log n.

Observe that several local minima (or maxima) may form a "chain" of successive vertices in G. Requirement (i), in the definition of an r-ruling set (in Section 3), does not allow us to include all these local minima in the set of selected vertices. Our algorithm exploits the alternation property (defined below) of vector $SERIAL_1$ to overcome this problem.

The alternation property: Let i be a vertex and j be its successor. If bit number $SERIAL_1(i)$ of $SERIAL_0(i)$ is 0 (resp. 1), then this bit is 1 (resp. 0) in $SERIAL_0(j)$. (For $SERIAL_1(i)$ is the index of the rightmost bit on which $SERIAL_0(i)$ and $SERIAL_0(j)$ differ.)

Suppose that $i_1, i_2 \cdots$ is a chain in G such that $SERIAL_1(i)$ is a local minimum (resp. maximum) for every i in the chain. Then:

Fact 3: For all vertices in the chain $SERIAL_1$ is the same (i.e., $SERIAL_1(i_1) = SERIAL_1(i_2) = \cdots$). (By definition of local minimum).

Below, we consider bit number $SERIAL_1(i_1)$ of $SERIAL_0$ for all vertices in the chain.

Fact 4: The following sequence of bits is an alternating sequence of zeros and ones.

Bit number $SERIAL_1(i_1)$ of $SERIAL_0(i_1)$, bit number $SERIAL_1(i_2)$ $(= SERIAL_1(i_1))$ of $SERIAL_0(i_2)$, ..., bit number $SERIAL_1(i_j)$ $(= SERIAL_1(i_1))$ of $SERIAL_0(i_j)$, ...

(This is readily implied by the alternation property.)

We can now understand why we called our technique deterministic coin tossing. We associated zeros and ones with the vertices, based on their original serial numbers; these serial numbers were set deterministically. This association allows us to treat (apparently) similar vertices differently. Finally, note that coin tossing can be used for similar purposes.

We return to the algorithm. We select the following subset of vertices.

We select the last vertex in the list; we also select the first vertex if there are more than two vertices in the list.

We say an unselected vertex is *available* if neither of its neighbors was selected and it is a local minimum. We select all available vertices i that satisfy one of the following two properties.

(1) Neither of i's neighbors is available.

(2) Bit number $SERIAL_1(i)$ is 1.

Now, we say an unselected vertex is *available* if neither of its neighbors was selected and it is a local maximum. We select all available vertices i that satisfy one of the following two properties.

(1) Neither of i's neighbors is available.

(2) Bit number $SERIAL_1(i)$ is 1.

The selected vertices form a log n-ruling set. Requirement (i) is satisfied since we never select two adjacent vertices. Requirement (ii) is satisfied by Fact 2 and since every local extremum is either selected or is a neighbor of a vertex that was selected.

We have shown:

Theorem: A logn-ruling set can be obtained in O(1) time using n processors.

NYU COMPSCI TR-244 c.1
Cole, Richard
Approximate parallel
scheduling. Part I